**2019-09-16**

# 1 Finding and fixing floating point problems

Floating point arithmetic is not the same as real arithmetic. Even simple properties like associativity or distributivity of addition and multiplication only hold approximately. Thus, some computations that look fine in exact arithmetic can produce bad answers in floating point. What follows is a (very incomplete) list of some of the ways in which programmers can go awry with careless floating point programming.

## 1.1 Cancellation

If $\hat{x} = x(1+\delta_1)$ and $\hat{y} = y(1+\delta_2)$ are floating point approximations to $x$ and $y$ that are very close, then $\mathrm{fl}(\hat{x} - \hat{y})$ may be a poor approximation to $x - y$ due to *cancellation*. In some ways, the subtraction is blameless in this tail: if $x$ and $y$ are close, then $\mathrm{fl}(\hat{x} - \hat{y}) = \hat{x} - \hat{y}$, and the subtraction causes no additional rounding error. Rather, the problem is with the approximation error already present in $\hat{x}$ and $\hat{y}$.

The standard example of loss of accuracy revealed through cancellation is in the computation of the smaller root of a quadratic using the quadratic formula, e.g.

$$x = 1 - \sqrt{1-z}$$

for $z$ small. Fortunately, some algebraic manipulation gives an equivalent formula that does not suffer cancellation:

$$x = \left(1 - \sqrt{1-z}\right)\left(\frac{1+\sqrt{1-z}}{1+\sqrt{1-z}}\right) = \frac{z}{1+\sqrt{1-z}}.$$

## 1.2 Sensitive subproblems

We often solve problems by breaking them into simpler subproblems. Unfortunately, it is easy to produce badly-conditioned subproblems as steps to solving a well-conditioned problem. As a simple (if contrived) example, try running the following MATLAB code:

```
x = 2;
for k = 1:60, x = sqrt(x); end
```

```
for k = 1:60, x = x^2; end
disp(x);
```

In exact arithmetic, this should produce 2, but what does it produce in floating point? In fact, the first loop produces a correctly rounded result, but the second loop represents the function $x^{2^{60}}$, which has a condition number far greater than $10^{16}$ — and so all accuracy is lost.

## 1.3 Unstable recurrences

One of my favorite examples of this problem is the recurrence relation for computing the integrals

$$E_n = \int_0^1 x^n e^{x-1}\, dx.$$

Integration by parts yields the recurrence

$$\begin{aligned} E_0 &= 1 - 1/e \\ E_n &= 1 - nE_{n-1}, \quad n \geq 1. \end{aligned}$$

This looks benign enough at first glance: no single step of this recurrence causes the error to explode. But each step amplifies the error somewhat, resulting in an exponential growth in error[1].

## 1.4 Undetected underflow

In Bayesian statistics, one sometimes computes ratios of long products. These products may underflow individually, even when the final ratio is not far from one. In the best case, the products will grow so tiny that they underflow to zero, and the user may notice an infinity or NaN in the final result. In the worst case, the underflowed results will produce nonzero subnormal numbers with unexpectedly poor relative accuracy, and the final result will be wildly inaccurate with no warning except for the (often ignored) underflow flag.

[1] Part of the reason that I like this example is that one can run the recurrence *backward* to get very good results, based on the estimate $E_n \approx 1/(n+1)$ for $n$ large.

### 1.5 Bad branches

A NaN result is often a blessing in disguise: if you see an unexpected NaN, at least you *know* something has gone wrong! But all comparisons involving NaN are false, and so when a floating point result is used to compute a branch condition and an unexpected NaN appears, the result can wreak havoc. As an example, try out the following code in MATLAB.

```
x = 0/0;
if x < 0,    disp('x is negative');
elseif x >= 0, disp('x is non-negative');
else         disp('Uh...');
end
```

## 2 Sums and dots

We already described a couple of floating point examples that involve evaluation of a fixed formula (e.g. computation of the roots of a quadratic). We now turn to the analysis of some of the building blocks for linear algebraic computations: sums and dot products.

### 2.1 Sums two ways

As an example of first-order error analysis, consider the following code to compute a sum of the entries of a vector $v$:

```
  s = 0;
  for k = 1:n
    s = s + v(k);
  end
```

Let $\hat{s}_k$ denote the computed sum at step $k$ of the loop; then we have

$$\begin{aligned}\hat{s}_1 &= v_1 \\ \hat{s}_k &= (\hat{s}_{k-1} + v_k)(1+\delta_k), \quad k > 1.\end{aligned}$$

Running this forward gives

$$\begin{aligned}\hat{s}_2 &= (v_1 + v_2)(1+\delta_2) \\ \hat{s}_3 &= ((v_1 + v_2)(1+\delta_2) + v_3)(1+\delta_2)\end{aligned}$$

and so on. Using first-order analysis, we have

$$\hat{s}_k \approx (v_1 + v_2)\left(1 + \sum_{j=2}^{k} \delta_j\right) + \sum_{l=3}^{k} v_l \left(1 + \sum_{j=l}^{k} \delta_j\right),$$

and the difference between $\hat{s}_k$ and the exact partial sum is then

$$\hat{s}_k - s_k \approx \sum_{j=2}^{k} s_j \delta_j.$$

Using $\|v\|_1$ as a uniform bound on all the partial sums, we have

$$|\hat{s}_n - s_n| \lesssim (n-1)\epsilon_{\text{mach}} \|v\|_2.$$

An alternate analysis, which is a useful prelude to analyses to come involves writing an error recurrence. Taking the difference between $\hat{s}_k$ and the true partial sums $s_k$, we have

$$\begin{aligned} e_1 &= 0 \\ e_k &= \hat{s}_k - s_k \\ &= (\hat{s}_{k-1} + v_k)(1 + \delta_k) - (s_{k-1} + v_k) \\ &= e_{k-1} + (\hat{s}_{k-1} + v_k)\delta_k, \end{aligned}$$

and $\hat{s}_{k-1} + v_k = s_k + O(\epsilon_{\text{mach}})$, so that

$$|e_k| \leq |e_{k-1}| + |s_k|\epsilon_{\text{mach}} + O(\epsilon_{\text{mach}}^2).$$

Therefore,

$$|e_n| \lesssim (n-1)\epsilon_{\text{mach}} \|v\|_1,$$

which is the same bound we had before.

## 2.2 Backward error analysis for sums

In the previous subsection, we showed an error analysis for partial sums leading to the expression:

$$\hat{s}_n \approx (v_1 + v_2)\left(1 + \sum_{j=2}^{n} \delta_j\right) + \sum_{l=3}^{n} v_l \left(1 + \sum_{j=l}^{n} \delta_j\right).$$

We then proceded to aggregate all the rounding error terms in order to estimate the error overall. As an alternative to aggregating the roundoff, we can also treat the rounding errors as perturbations to the input variables (the entries of $v$); that is, we write the computed sum as

$$\hat{s}_n = \sum_{j=1}^{n} \hat{v}_j$$

where

$$\hat{v}_j = v_j(1+\eta_j), \qquad \text{where } |\eta_j| \lesssim (n+1-j)\epsilon_{\text{mach}}.$$

This gives us a *backward error* formulation of the rounding: we have re-cast the role of rounding error in terms of a perturbation to the input vector $v$. In terms of the 1-norm, we have the relative error bound

$$\|\hat{v} - v\|_1 \lesssim n\epsilon_{\text{mach}}\|v\|_1;$$

or we can replace $n$ with $n-1$ by being a little more careful. Either way, what we have shown is that the summation algorithm is *backward stable*, i.e. we can ascribe the roundoff to a (normwise) small relative error with a bound of $C\epsilon_{\text{mach}}$ where the constant $C$ depends on the size $n$ like some low-degree polynomial.

Once we have a bound on the backward error, we can bound the forward error via a condition number. That is, suppose we write the true and perturbed sums as

$$s = \sum_{j=1}^{n} v_j \qquad\qquad \hat{s} = \sum_{j=1}^{n} \hat{v}_j.$$

We want to know the relative error in $\hat{s}$ via a normwise relative error bound in $\hat{v}$, which we can write as

$$\frac{|\hat{s}-s|}{|s|} = \frac{|\sum_{j=1}^{n}(\hat{v}_j - v_j)|}{|s|} \leq \frac{\|\hat{v}-v\|_1}{|s|} = \frac{\|v\|_1}{|s|}\frac{\|\hat{v}-v\|_1}{\|v\|_1}.$$

That is, $\|v\|_1/|s|$ is the condition number for the summation problem, and our backward stability analysis implies

$$\frac{|\hat{s}-s|}{|s|} \leq \frac{\|v\|_1}{|s|} n\epsilon_{\text{mach}}.$$

This is the general pattern we will see again in the future: our analysis consists of a backward error computation that depends purely on the algorithm, together with a condition number that depends purely on the problem. Together, these give us forward error bounds.

## 2.3 Running error bounds for sums

In all the analysis of summation we have done so far, we ultimately simplified our formulas by bounding some quantity in terms of $\|v\|_1$. This is nice for algebra, but we lose some precision in the process. An alternative is to compute a *running error bound*, i.e. augment the original calculation with something that keeps track of the error estimates. We have already seen that the error in the computations looks like

$$\hat{s}_n - s_n = \sum_{j=2}^{n} s_j \delta_j + O(\epsilon_{\text{mach}}^2),$$

and since $s_j$ and $\hat{s}_j$ differ only by $O(\epsilon_{\text{mach}})$ terms,

$$|\hat{s}_n - s_n| \lesssim \sum_{j=2}^{n} |\hat{s}_j| \epsilon_{\text{mach}} + O(\epsilon_{\text{mach}}^2),$$

We are not worried about doing a rounding error analysis of our rounding error analysis — in general, we care more about order of magnitude for rounding error anyhow — so the following routine does an adequate job of computing an (approximate) upper bound on the error in the summation:

```
s = 0;
e = 0;
for k = 1:n
  s = s + v(k);
  e = e + abs(s) * eps;
end
```

## 2.4 Compensated summation

We conclude our discussion of rounding analysis for summation with a comment on the *compensated summation* algorithm of Kahan, which is not amenable to straightforward $1 + \delta$ analysis. The algorithm maintains the partial sums not as a single variable `s`, but as an unevaluated sum of two variables `s` and `c`:

```
s = 0;
c = 0;
for k = 1:n
  y = v(i) - c;
  t = s + y;
  c = (t - s) - y; % Key step
  s = t;
end
```

Where the error bound for ordinary summation is $(n-1)\epsilon_{\text{mach}}\|v\|_1 + O(\epsilon_{\text{mach}}^2)$, the error bound for compensated summation is $2\epsilon_{\text{mach}}\|v\|_1 + O(\epsilon_{\text{mach}}^2)$. Moreover, compensated summation is exact for adding up to $2^k$ terms that are within about $2^{p-k}$ of each other in magnitude.

Nor is Kahan's algorithm the end of the story! Higham's *Accuracy and Stability of Numerical Methods* devotes an entire chapter to summation methods, and there continue to be papers written on the topic. For our purposes, though, we will wrap up here with two observations:

- Our initial analysis in the $1+\delta$ model illustrates the general shape these types of analyses take and how we can re-cast the effect of rounding errors as a "backward error" that perturbs the inputs to an exact problem.
- The existence of algorithms like Kahan's compensated summation method should indicate that the backward-error-and-conditioning approach to rounding analysis is hardly the end of the story. One could argue it is hardly the beginning! But it is the approach we will be using for most of the class.

## 2.5 Dot products

We conclude with one more example error analysis, this time involving a real dot product computed by a loop of the form

```
dot = 0;
for k = 1:n
  dot = dot + x(k)*y(k);
end
```

Unlike the simple summation we analyzed above, the dot product involves two different sources of rounding errors: one from the summation, and one from the product. As in the case of simple summations, it is convenient to

re-cast this error in terms of perturbations to the input. We could do this all in one go, but since we have already spent so much time on summation, let us instead do it in two steps. Let $v_k = x_k y_k$; in floating point, we get $\hat{v}_k = v_k(1+\eta_k)$ where $|\eta_k| < \epsilon_{\text{mach}}$. Further, we have already done a backward error analysis of summation to show that the additional error in summation can be cast onto the summands, i.e. the floating point result is $\sum_k \tilde{v}_k$ where

$$\begin{aligned}\tilde{v}_k &= \hat{v}_k(1 + \sum_{j=\min(2,n)}^{n} \delta_j)(1+\eta_k) + O(\epsilon_{\text{mach}}^2) \\ &= v_k(1+\gamma_k) + O(\epsilon_{\text{mach}}^2)\end{aligned}$$

where

$$|\gamma_k| = |\eta_k + \sum_{j=\min(2,n)}^{n} \delta_j| \leq n\epsilon_{\text{mach}}.$$

Rewriting $v_k(1+\gamma_k)$ as $\hat{x}_k y_k$ where $\hat{x}_k = x_k(1+\gamma_k)$, we have that the computed inner product $y^T x$ is equivalent to the exact inner product of $y^T \hat{x}$ where $\hat{x}$ is an elementwise relatively accurate (to within $n\epsilon_{\text{mach}} + O(\epsilon_{\text{mach}}^2)$) approximation to $x$.

A similar backward error analysis shows that computed matrix-matrix products $AB$ in general can be interpreted as $\hat{A}B$ where

$$|\hat{A} - A| < p\epsilon_{\text{mach}}|A| + O(\epsilon_{\text{mach}}^2)$$

and $p$ is the inner dimension of the product. Exactly what $\hat{A}$ is depends not only on the data, but also the loop order used in the multiply — since, as we recall, the order of accumulation may vary from machine to machine depending on what blocking is best suited to the cache! But the bound on the backward error holds for all the common re-ordering[2] And this backward error characterization, together with the type of sensitivity analysis for matrix multiplication that we have already discussed, gives us a uniform framework for obtaining forward error bounds for matrix-matrix muliplication; and the same type of analysis will continue to dominate our discussion of rounding errors as we move on to more complicated matrix computations.

[2]For those of you who know about Strassen's algorithm — it's not backward stable, alas.

# 3 Problems to ponder

1. How do we accurately evaluate $\sqrt{1+x} - \sqrt{1-x}$ when $x \ll 1$?
2. How do we accurately evaluate $\ln\sqrt{x+1} - \ln\sqrt{x}$ when $x \gg 1$?
3. How do we accurately evaluate $(1-\cos(x))/\sin(x)$ when $x \ll 1$?
4. How would we compute $\cos(x) - 1$ accurately when $x \ll 1$?
5. The *Lamb-Oseen vortex* is a solution to the 2D Navier-Stokes equation that plays a key role in some methods for computational fluid dynamics. It has the form

   $$v_\theta(r,t) = \frac{\Gamma}{2\pi r}\left(1 - \exp\left(\frac{-r^2}{4\nu t}\right)\right)$$

   How would one evaluate $v(r,t)$ to high relative accuracy for all values of $r$ and $t$ (barring overflow or underflow)?
6. For $x > 1$, the equation $x = \cosh(y)$ can be solved as

   $$y = -\ln\left(x - \sqrt{x^2-1}\right).$$

   What happens when $x = 10^8$? Can we fix it?
7. The difference equation

   $$x_{k+1} = 2.25x_k - 0.5x_{k-1}$$

   with starting values

   $$x_1 = \frac{1}{3}, \qquad x_2 = \frac{1}{12}$$

   has solution

   $$x_k = \frac{4^{1-k}}{3}.$$

   Is this what you actually see if you compute? What goes wrong?
8. Considering the following two MATLAB fragments:

```
% Version 1
f = (exp(x)-1)/x;

% Version 2
y = exp(x);
f = (1-y)/log(y);
```

In exact arithmetic, the two fragments are equivalent. In floating point, the first formulation is inaccurate for $x \ll 1$, while the second formulation remains accurate. Why?

9. Running the recurrence $E_n = 1 - nE_{n-1}$ *forward* is an unstable way to compute $\int_0^1 x^n e^{x-1}\,dx$. However, we can get good results by running the recurrence *backward* from the estimate $E_n \approx 1/(N+1)$ starting at large enough $N$. Explain why. How large must $N$ be to compute $E_{20}$ to near machine precision?

10. How might you accurately compute this function for $|x| < 1$?

$$f(x) = \sum_{j=0}^{\infty} \left(\cos(x^j) - 1\right)$$